StochJuMP – Parallel algebraic modelling for stochastic optimization in Julia

Cosmin G. Petra

Argonne National Laboratory

petra@mcs.anl.gov

Joey Huchette, Miles Lubin

MIT Operations Research Center

PASC15 Conference

June 1, 2015

Outline

- Motivating applications: energy applications

- Parallel optimization solvers for stochastic optimization
  - PIPS solvers suite @ Argonne
  - Computational pattern of stochastic optimization

- Modelling stochastic optimization on HPC platforms
  - JuMP – algebraic modelling language for optimization embedded in Julia
  - StochJuMP extension of JuMP for parallel modelling
  - Technical details and numerical experiments

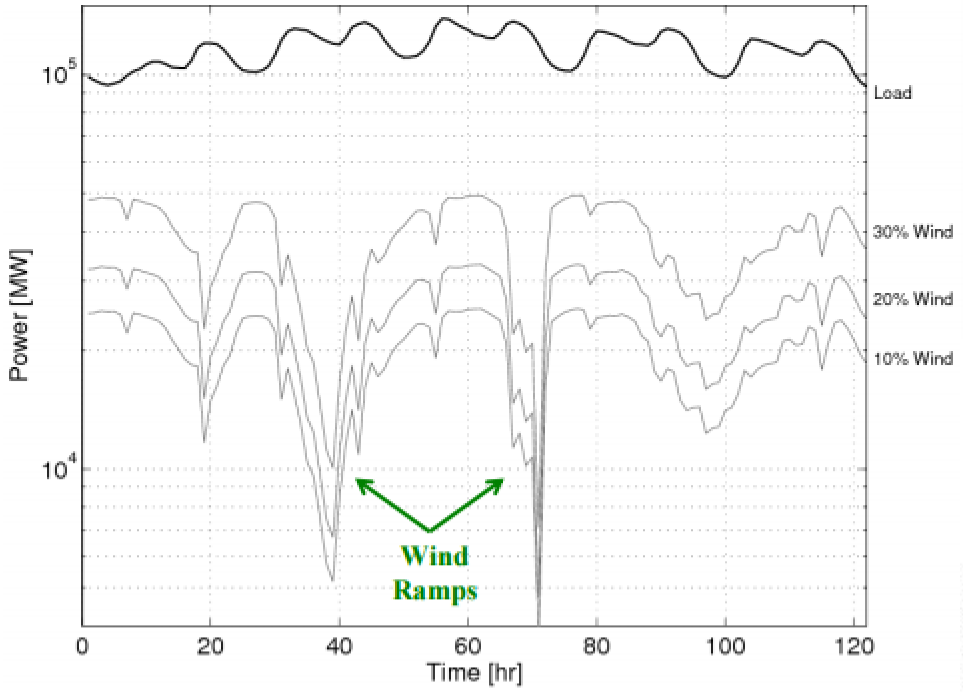
Stochastic optimization

- **Optimization under uncertainty**: take an optimal decision **now** that depends on future, **uncertain** events (random variable)

- **Stochastic optimization**: the “now” hedges against all possible realizations of the randomness (by minimizing the expectation of the cost).

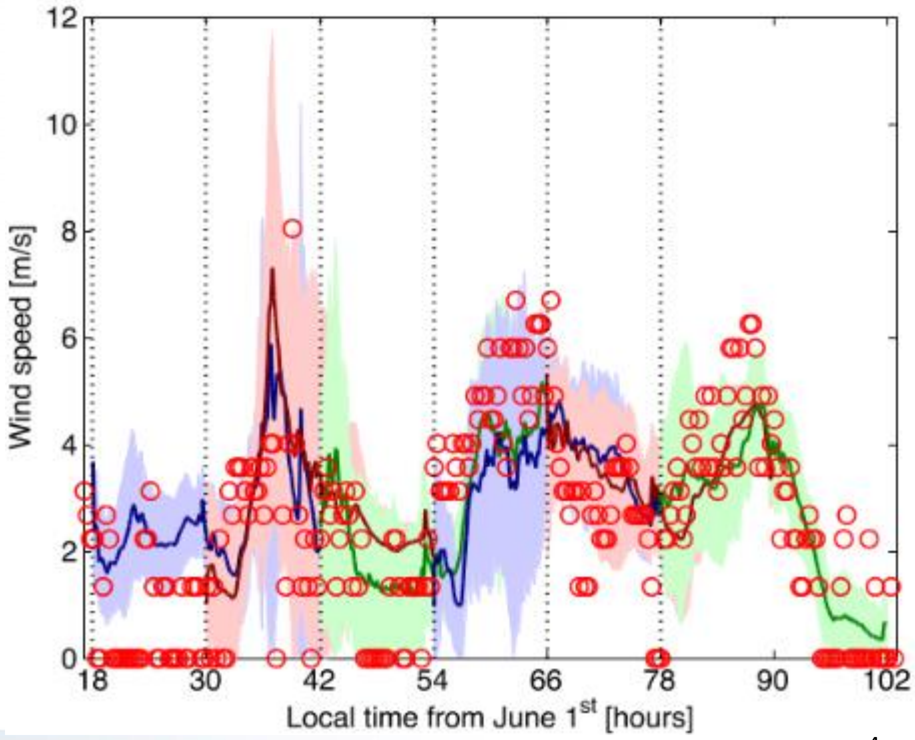
$$\begin{aligned} \min_{x_0} \quad & \mathbb{E}_\xi[f(x_0,\xi)] \\ \text{subj.to:} \quad & x_0 \in \Theta \end{aligned}$$

Electricity generation and dispatch under uncertainty



The sharp drops in wind power need to be forecasted well in advanced to give the thermal generators enough time to ramp up production.

Wind forecasting results in wind scenarios, requiring stochastic optimization



Two-stage stochastic programming with recourse

$$\begin{aligned}
&\min_x c^T x + \mathbb{E}_\xi[G(x,\xi)] \\
&\text{s.t. } Ax = b, x \geq 0,
\end{aligned}$$

where the *recourse function* $G(x,\xi)$ is defined by

$$\begin{aligned}
G(x,\xi) = &\min_y c_\xi^T y \\
&\text{s.t. } T_\xi x + W_\xi y = b_\xi, y \geq 0.
\end{aligned}$$

- Wide range of applications
- In energy: power grid/ natural gas operations, N-1 contingency analysis, generation expansion, transmission planning, etc

Wind/solar (stochastic) economic dispatch model

$$\begin{aligned}
\min_{x,X(\omega),f,F(\omega)} \quad & \sum_{i\in G} c_i x_i + \mathbb{E}_\omega \sum_{i\in G} p_i |x_i - y_i(\omega)| \\
\text{subj.to:} \quad & \tau_n(f) + \sum_{i\in T(n)} x_i = d_n, \forall n \in N \\
& \tau_n(F(\omega)) + \sum_{i\in T(n)} y_i(\omega) = d_n, \forall n \in N, \omega \in \Omega \\
& f, F(\omega) \in U, \forall \omega \in \Omega \\
& x_i, y_i(\omega) \in U_i, \forall i \in G, \omega \in \Omega
\end{aligned}$$

Ω is the set of **weather scenarios**

x_i	-	generation
f	-	vector of line flows
$\tau_n(f)$	-	net power imported at node n
N	-	set of network nodes
d_n	-	demand at node n
$T(n)$	-	set of generators at node n
U	-	line capacity constraints
G	-	set of generators (coal, gas, nuclear, wind, solar)
C_i	-	generation constraints
c_i	-	generation costs

Large-scale (dual) block-angular LPs

After taking a finite samples, problem reduces to a large deterministic problem known as extensive form

$$
\begin{array}{lllllll}
\min & c_0^T x + \sum_{i=1}^N c_i^T y_i & & & & & \\
\text{s.t.} & Ax & & & & & = b_0, \\
 & T_1 x + & W_1 y_1 & & & & = b_1, \\
 & T_2 x + & & W_2 y_2 & & & = b_2, \\
 & \vdots & & & \ddots & & \vdots \\
 & T_N x + & & & & W_N y_N & = b_N, \\
 & x \geq 0, & y_1 \geq 0, & y_2 \geq 0, & \ldots, & y_N \geq 0. &
\end{array}
$$

Large instances with 1000s of scenarios could have billions of variables and constraints, requiring memory distributed parallel computing.

Parallel optimization solver(s)

PIPS solvers

- PIPS-IPM – stochastic LPs and convex QPs
  - Mehrotra predictor-corrector interior-point method (IPM)
- PIPS-S – dual block-angular LPs (includes stochastic LPs)
  - Parallel implementation of revised dual simplex
- PIPS-NLP – stochastic NLPs
  - Reuses PIPS-IPM linear algebra
  - Inertia-free filter method (Chiang and Zavala, 2014)
  - Various structure-exploiting implementations (network, PDEs, etc)
- Parallelization obtained at the linear algebra level

Parallel interior-point method implementation – PIPS-IPM

When using an interior-point method to solve the extensive form, the linear systems are **structured**

$$\begin{bmatrix} K_1 & & & B_1 \\ & \ddots & & \vdots \\ & & K_N & B_N \\ B_1^T & \dots & B_N^T & K_0 \end{bmatrix} \begin{bmatrix} \Delta z_1 \\ \vdots \\ \Delta z_N \\ \Delta z_0 \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_N \\ r_0 \end{bmatrix}$$

arrow-shaped linear systems

(modulo a permutation)

Schur complement decomposition of linear algebra

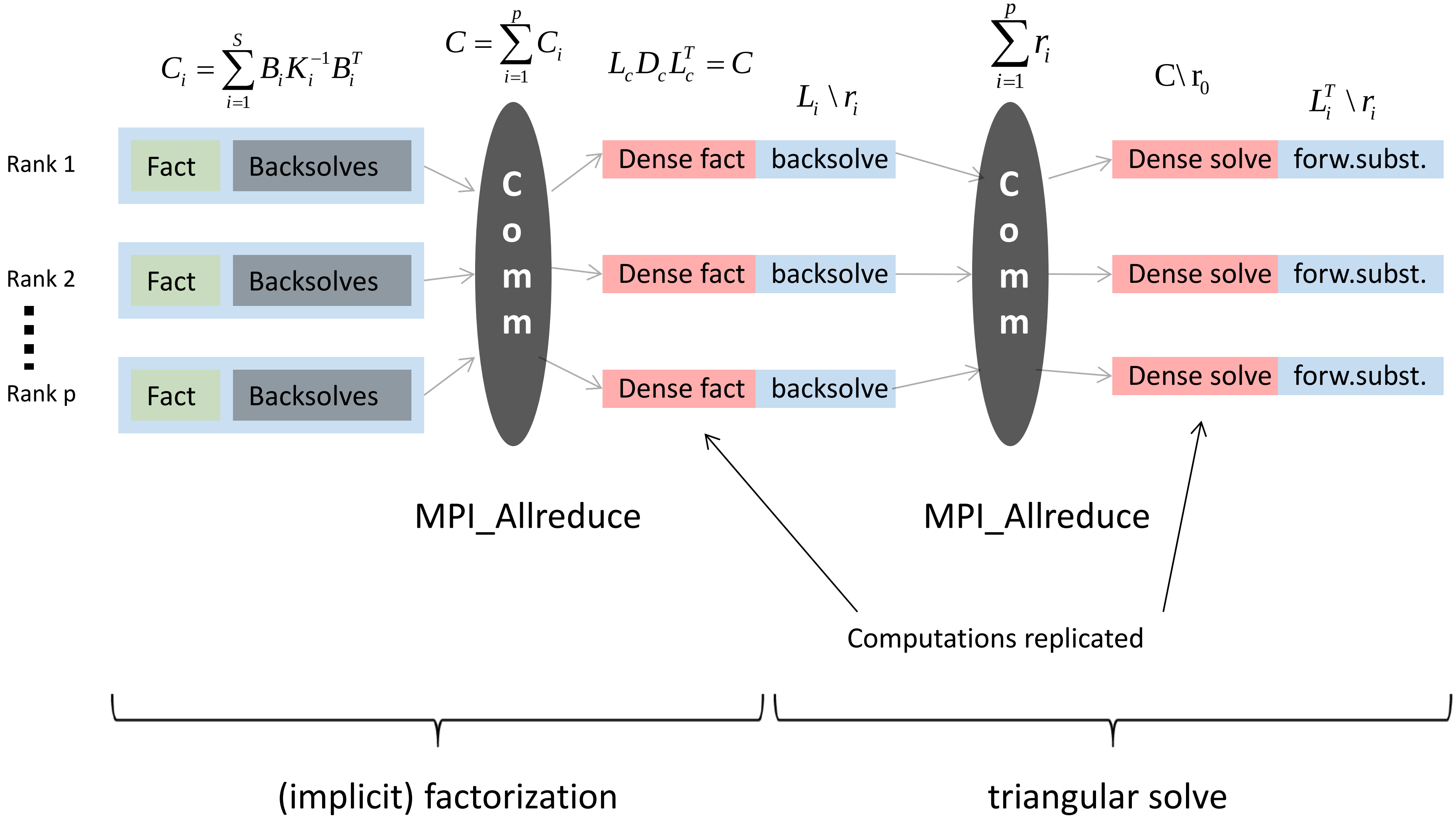
$$\begin{bmatrix} K_1 & & & B_1 \\ & \ddots & & \vdots \\ & & K_N & B_N \\ B_1^T & \dots & B_N^T & K_0 \end{bmatrix} \begin{bmatrix} \Delta z_1 \\ \vdots \\ \Delta z_N \\ \Delta z_0 \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_N \\ r_0 \end{bmatrix}$$

Block elimination

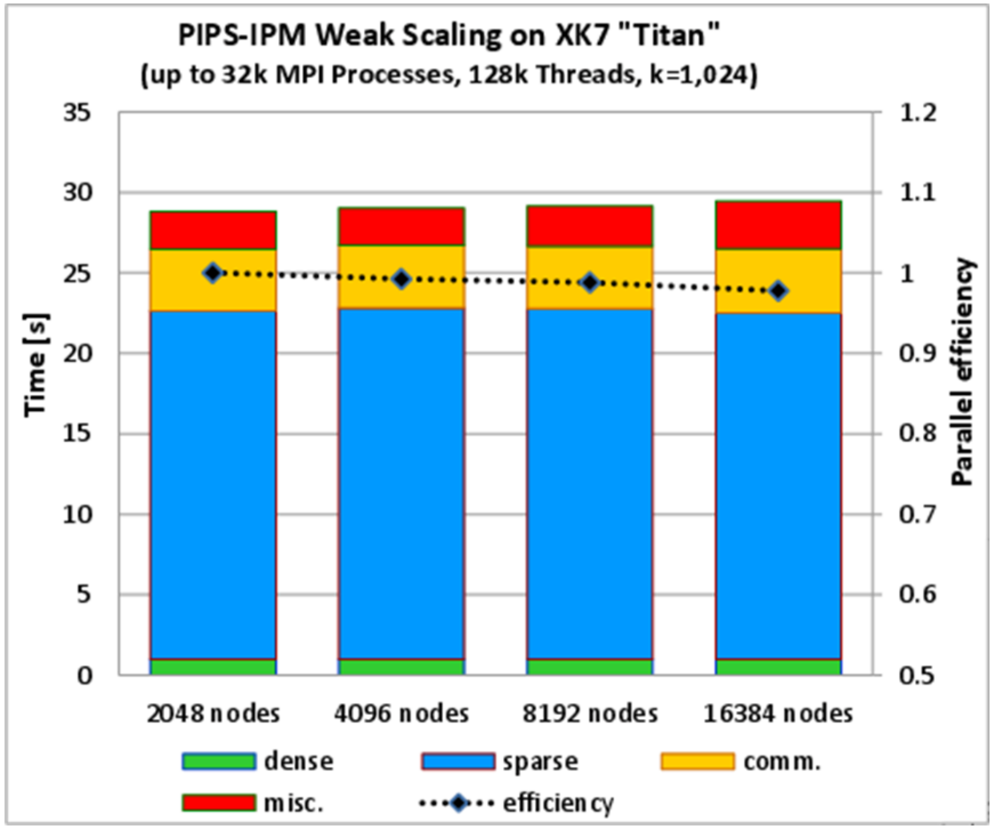
$$\left(K_0 - \sum_{i=1}^{N} B_i^T K_i^{-1} B_i \right) \Delta z_0 = r_0 - \sum_{i=1}^{N} B_i^T K_i^{-1} r_i$$

The matrix $C := K_0 - \sum_{i=1}^{N} B_i^T K_i^{-1} B_i$ is the Schur-complement of the diagonal $K_1, \dots, K_N$ block.

Parallel computational pattern

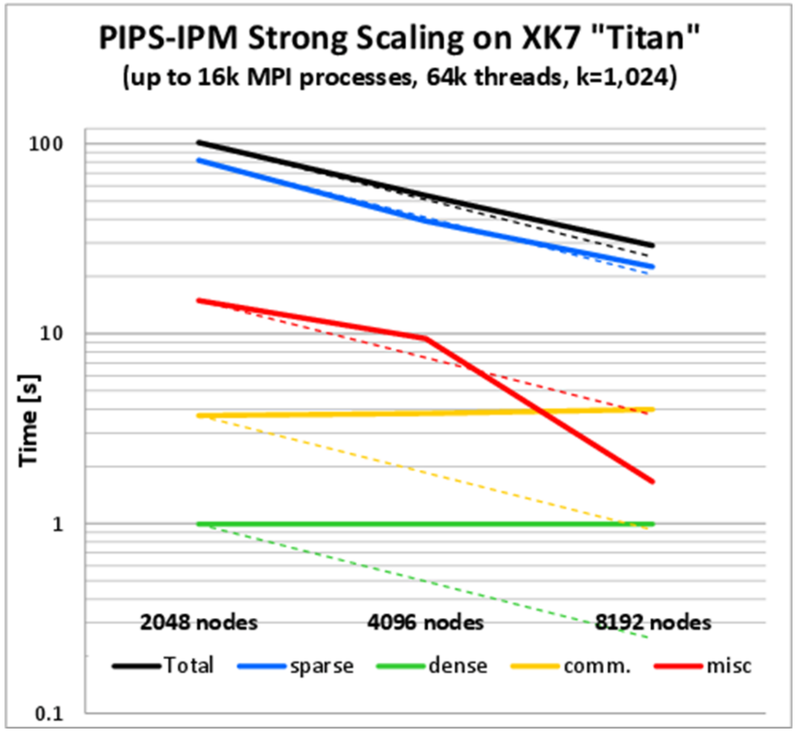


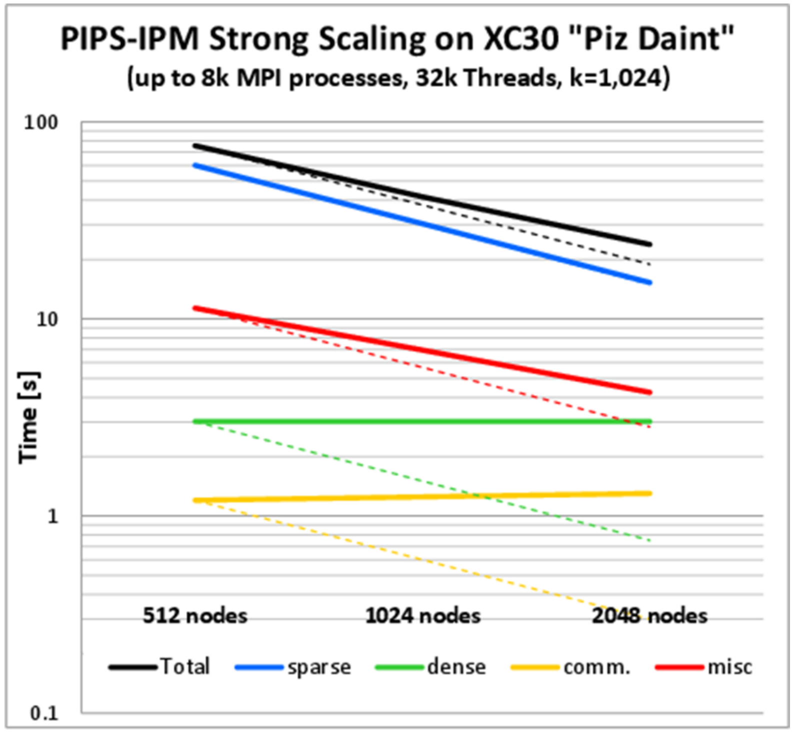
Weak scaling efficiency – Titan @ Oak Ridge National Lab



Largest instance has 4.08 billion decision variables and 4.12 billion constraints.

Strong scaling – Titan and "Piz Daint" (@ Swiss National Computing Center)





The instance used in the XK7 runs has 4.08 billion decision variables and 4.12 billion constraints.

Structure-exploiting solvers generally scale.

How about modelling?

Modelling structured optimization problems on high performance computing (HPC) platforms

- Algebraic modelling language/framework
  - easy-to-express syntax, similar to the mathematical abstractions
  - “high performance”
    - scalable and efficient models generation in parallel (data distributed and localized)
    - code speed – ideally C/Fortran speed
    - minimum I/O
  - transparently passes structure to the optimization solver
  - quick development; easy to specialize and/or extend
  - plug-and-play with optimization solvers (generally Fortran, C, C++ codes)

- Existing modelling frameworks with parallel capabilities: SML (Grothey et al., 2009), PySP (Watson et al, 2012), PSMG (Qiang and Grothey, 2014)

Our approach is to extend JuMP

- JuMP - open-source algebraic modeling language for mathematical programming embedded in Julia (Miles Lubin, Iain Dunning, Joey Huchette – MIT)

- Solver-independent, extensible, domain-specific language with “optimization syntax”

- JuMP exploits advanced language features of Julia
  - Metaprogramming, not operator overloading
  - Just-in-time compilation
  - Excellent connections to C/Fortran libraries
  - Optional typing, multiple dispatch

JuMP's expressiveness and speed

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} p_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{N} w_i x_i \leq C \\ & 0 \leq x \leq 1 \end{aligned}$$

```
m = Model(:Max)
@defVar(m, 0 <= x[j=1:N] <= 1)
@setObjective(m, sum{profit[j] * x[j], j=1:N})
@addConstraint(m, sum{weight[j] * x[j], j = 1:N} <= C)
```

Table: Linear-quadratic control benchmark results. N=M is the grid size. Total time (in seconds) to process the model definition and produce the output file in LP and MPS formats (as available).

	JuMP/Julia		AMPL	Gurobi/C++		Pulp/PyPy		Pyomo
N	LP	MPS	MPS	LP	MPS	LP	MPS	LP
250	0.5	0.9	0.8	1.2	1.1	8.3	7.2	13.3
500	2.0	3.6	3.0	4.5	4.4	27.6	24.4	53.4
750	5.0	8.4	6.7	10.2	10.1	61.0	54.5	121.0
1,000	9.2	15.5	11.6	17.6	17.3	108.2	97.5	214.7

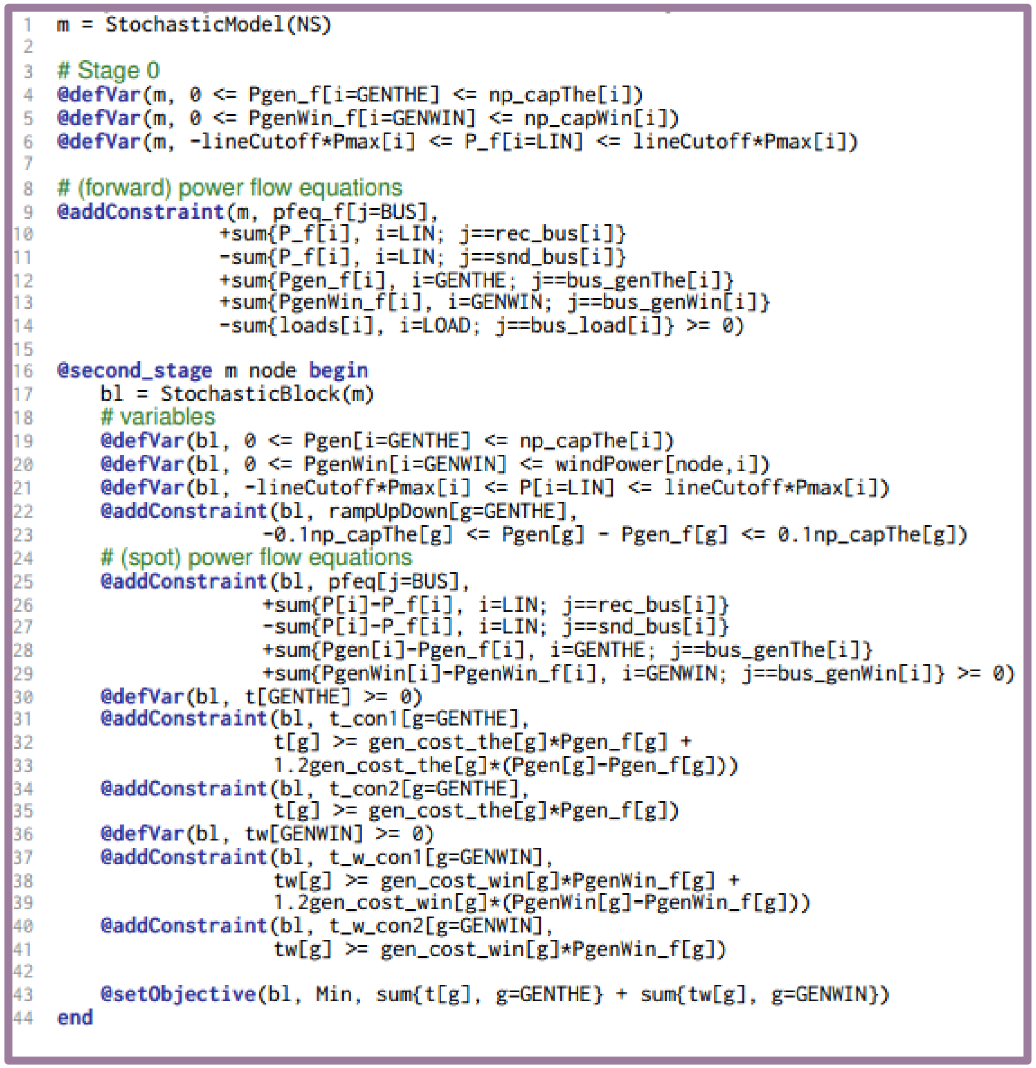
StochJuMP - parallel algebraic modeling for stochastic optimization

- Technical approach: built as an extention on top of JuMP – very little extra code
- Uses JuMP’s extension system to reuse data structures (and code!)
  - Each scenario subproblem is a JuMP model

- Minimal language constructs: **StochasticModel**, **StochasticBlock**

- Generic: usable with any solver, given backend glue code

The full stochastic economic dispatch model for the State of Illinois



```
m = StochasticModel(NS)

# Stage 0
@defVar(m, 0 <= Pgen_f[i=GENTHE] <= np_capThe[i])
@defVar(m, 0 <= PgenWin_f[i=GENWIN] <= np_capWin[i])
@defVar(m, -lineCutoff*Pmax[i] <= P_f[i=LIN] <= lineCutoff*Pmax[i])

# (forward) power flow equations
@addConstraint(m, pfeq_f[j=BUS],
              +sum{P_f[i], i=LIN; j==rec_bus[i]}
              -sum{P_f[i], i=LIN; j==snd_bus[i]}
              +sum{Pgen_f[i], i=GENTHE; j==bus_genThe[i]}
              +sum{PgenWin_f[i], i=GENWIN; j==bus_genWin[i]}
              -sum{loads[i], i=LOAD; j==bus_load[i]} >= 0)

@second_stage m node begin
    bl = StochasticBlock(m)
    # variables
    @defVar(bl, 0 <= Pgen[i=GENTHE] <= np_capThe[i])
    @defVar(bl, 0 <= PgenWin[i=GENWIN] <= windPower[node,i])
    @defVar(bl, -lineCutoff*Pmax[i] <= P[i=LIN] <= lineCutoff*Pmax[i])
    @addConstraint(bl, rampUpDown[g=GENTHE],
                 -0.1np_capThe[g] <= Pgen[g] - Pgen_f[g] <= 0.1np_capThe[g])
    # (spot) power flow equations
    @addConstraint(bl, pfeq[j=BUS],
                 +sum{P[i]-P_f[i], i=LIN; j==rec_bus[i]}
                 -sum{P[i]-P_f[i], i=LIN; j==snd_bus[i]}
                 +sum{Pgen[i]-Pgen_f[i], i=GENTHE; j==bus_genThe[i]}
                 +sum{PgenWin[i]-PgenWin_f[i], i=GENWIN; j==bus_genWin[i]} >= 0)
    @defVar(bl, t[GENTHE] >= 0)
    @addConstraint(bl, t_con1[g=GENTHE],
                  t[g] >= gen_cost_the[g]*Pgen_f[g] +
                  1.2gen_cost_the[g]*(Pgen[g]-Pgen_f[g]))
    @addConstraint(bl, t_con2[g=GENTHE],
                  t[g] >= gen_cost_the[g]*Pgen_f[g])
    @defVar(bl, tw[GENWIN] >= 0)
    @addConstraint(bl, t_w_con1[g=GENWIN],
                  tw[g] >= gen_cost_win[g]*PgenWin_f[g] +
                  1.2gen_cost_win[g]*(PgenWin[g]-PgenWin_f[g]))
    @addConstraint(bl, t_w_con2[g=GENWIN],
                  tw[g] >= gen_cost_win[g]*PgenWin_f[g])

    @setObjective(bl, Min, sum{t[g], g=GENTHE} + sum{tw[g], g=GENWIN})
end
```

Parallel model generation and interfacing with PIPS-IPM

- Data is localized: processes only generate data for scenarios assigned to them.

1. Convert abstract JuMP model to problem data (before calling out to PIPS-IPM)
2. Construct thin Julia wrapper functions to copy local data to PIPS buffers
3. Initialize PIPS-IPM and provide the MPI communicator
4. Pass “C” functions to PIPS-IPM via **cfunction**
5. Call PIPS-IPM solve function
6. Post-solve analysis (in Julia)

That’s it! No Magic.

- 300 lines of Julia code
- 2 weeks of work, but only because the 2014 World Cup was in progress

Computational results

Modelling also scales.

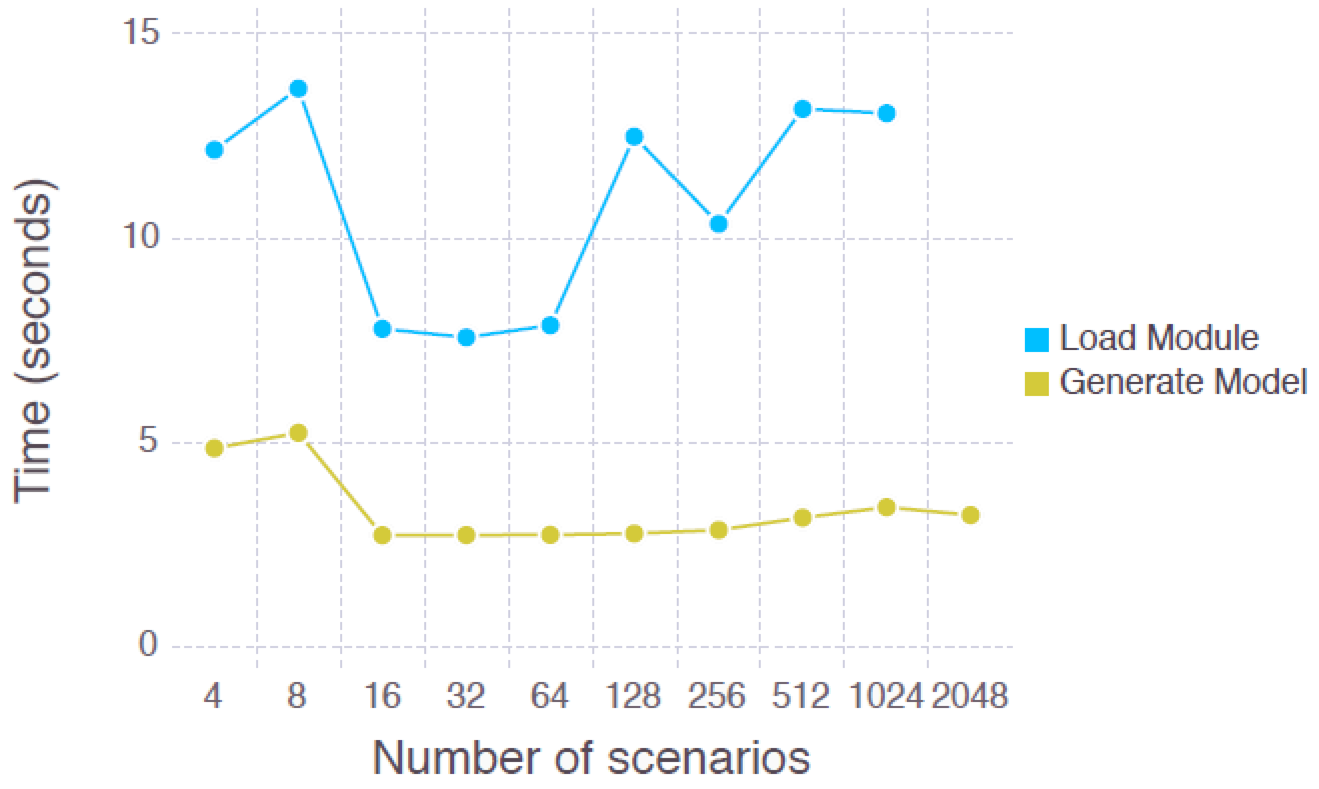
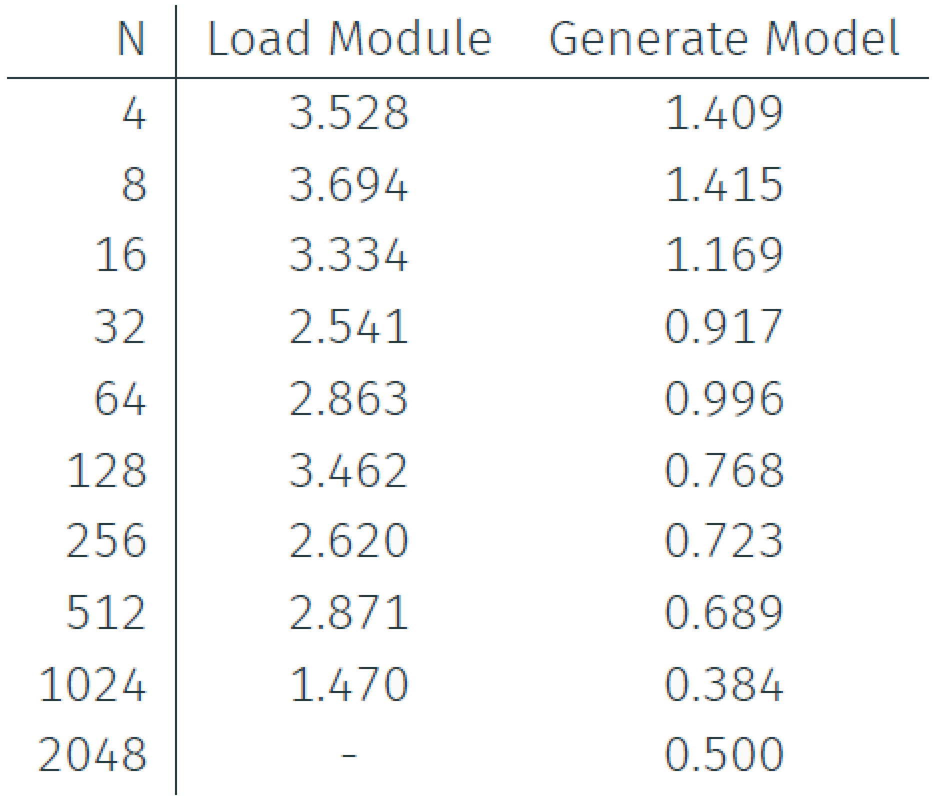


Figure: Weak scaling study from 4 to 2048 cores

Computational results - continued

Model generation always less than 1.5% of solve time (and typically less)



N	Load Module	Generate Model
4	3.528	1.409
8	3.694	1.415
16	3.334	1.169
32	2.541	0.917
64	2.863	0.996
128	3.462	0.768
256	2.620	0.723
512	2.871	0.689
1024	1.470	0.384
2048	-	0.500

Table: Ratio of StochJuMP timings over solve time ($\times 100$).

Future work

- Extending StochJuMP to nonlinear stochastic optimization
  - Automatic differentiation is needed (already in place in JuMP)

- Develop other domain specific (modelling) languages in Julia/JuMP
  - dynamic optimization (a.k.a optimal control)

Thank you for your attention!

Questions?

Additional slides

Economic dispatch models

- Basis for the electricity distribution and electricity market

$$\begin{aligned} \min_{x,f} \quad & \sum_{i \in G} c_i x_i \\ \text{subj.to:} \quad & \tau_n(f) + \sum_{i \in T(n)} x_i = d_n, \forall n \in N \\ & f \in U \\ & x_i \in C_i, \forall i \in G \end{aligned}$$

x_i	-	generation
f	-	vector of line flows
$\tau_n(f)$	-	net power imported at node n
N	-	set of network nodes
d_n	-	demand at node n
$T(n)$	-	set of generators at node n
U	-	line capacity constraints
G	-	set of generators (coal, gas, nuclear, wind, solar)
C_i	-	generation constraints
c_i	-	generation costs

- Answers critical questions such as:
  - What is the cheapest way to ramp-up generation to satisfy a foreseen increase in demand given the grid transmission limits imposed?
  - What are the electricity prices at each demand node given a certain demand?